The Token Test: Take II

Andrew Winterman

February 3, 2011

Assumptions, the General Idea, and Requisite Data Structures

My general idea is still to construct a big matrix, called X, with rows corresponding to lines of text, and columns corresponding to tokens. If the i^{th} line has the j^{th} token on it, then the ij^{th} entry of the matrix is flipped from 0 to 1. However, because we have no *a priori* knowledge of the length of the file and the token list, the matrix must be constructed in some efficient way as we read through the file. Once again, once the file has been read, we sum down the columns of X to find out how many times each token appeared. My assumptions are listed below:

1. There is no more than one token per line
2. Identifying tokens is not part of the problem. - i.e. they are clearly delineated within the data set.
3. It is inexpensive to read each line
4. We have no a priori knowledge of either the token list or the length of the document.

My algorithm will use a hash function, f, to map tokens to indices of columns of X. The hash table defining f will be changed as new tokens are discovered. Matching them to indices in the order they were discovered.

X will be a linked list of linked lists, to allow for easy on-the-spot modification. Each list in X corresponds to a token (column). When a new token is discovered, a linked list of form $\{0, 0, \ldots, 0, 1\}$ is added to X with length equal to the number of lines read so far. Each additional line read adds an entry to each list. At the end of the document, each list in X will be summed and printed with its corresponding token.

Algorithm in Psuedo-code

(First we initialize the matrix X and the hash function f)
Let T be the list of tokens discovered
Let $N_k = \{0, 1, \ldots, k\}$, where k is the number of tokens discovered.

Let $f : T \to N_k$ be defined as follows as described above. i.e. f(smallest token) = 1, f(second smallest token) = 2, and so on. T and N_k are global variables.

(f is the hash function, and sends the Token corresponding to the empty string to 0.)

LET $X = \{\{\}\}$,

(an empty linked list of linked lists)

LET $T = \{\}$ and $k = 0$

(Because we have not discovered any tokens, yet.)

LET n = 1

WHILE [Document has not ended]

read line n

IF [Token is found AND $f(\text{Token}) = k + 1$]

(i.e. if a new token is discovered on line n)

addend a list of form $\{0, 0, \ldots, 1\}$ and length n to the end of X

(the new list should be constructed by specifying that the last entry is 1, and that all preceding entries are 0.)

addend a '0' to all other lists in X.

LET $k = k + 1$, $N_k = \{1, 2, \ldots, k\}$, $T = T \cup \{\text{Token}\}$

IF [Token is found AND $f(\text{Token}) = j \leq k$]

(i.e. if an old token is found on line i)

addend a '1' to the j^{th} list of X

addend a '0' to all other lists in X.

ELSE

addend a '0' to the end of all lists in X

Increment n, the line index.

(At this point the whole document has been read)

FOR [j in $1, 2, \ldots, k$]

(recall that k is the total number unique tokens discovered)

Let $v_j = \sum_{j=1}^{n} x_j$

(where x_j is the j^{th} list in X.)

PRINT:

Tolken list | v

Time Complexity

Suppose the total number of tokens discovered is m, that the total number of lines in the document is n, and that the number of tokens discovered prior to the i^{th} iteration of the WHILE loop is k_i. In the worst case, each token is found only once in the document. In this case, the hash function takes $k_i - 1$ operations to discover the k_i^{th} token. The WHILE loop invokes the hash function at most m times, and then carries out k_i operations per time, So it looks like the while loop takes $O(n + m \cdot \sum_{i=1}^{m} 1)$ time, which is bounded by $O(n + m^2)$.

The FOR loop involves m summations of n terms, so if they are done sequentially it takes $m \cdot n$ operations. Assuming the time to read and identify a

token is independent of n and m, the total time complexity of the algorithm above is $O(m \cdot n + m^2)$. If we can compute all m sums simultaneously, then the algorithm runs in $O(n + m^2)$ time.

I do not believe this algorithm is optimal for its task. As a matter of fact, I think I could further optimize the hash function by ordering the tokens according to the character strings contained in them and using a binary search tree to hold all known tokens. Newly discovered token would be inserted into the appropriate place, as dictated by ordinality. This is unimplemented because I wanted to complete the analysis of the algorithm above before modifying it.

However, I have good reason to believe this would improve the time complexity of the problem, despite the added expense caused by modifying the binary search tree as new tokens are discovered. I found a paper[1] which states that binary search on a binary tree linked list takes log m time, if m is the length of the list. Hence implementing this change would improve the time complexity of the WHILE loop to $O(n + m \log m)$ because each call to the hash function would take log k_i (i = 1, ..., m) rather than $k_i - 1$ operations.

[1]Khosraviyani F., "Using Binary Search On a Linked List," published in Newsletter ACM SIGCSE Bulletin Homepage archive, Volume 22 Issue 3, Sep. 1990.